Cross-Review Report: Shell Sort vs. Heap Sort

Group: SE-2431, Boranbay Zere and Jangazy Bakytzhan

---

1. Algorithm Overview

Partner's Algorithm – Shell Sort

Shell Sort is an extension of Insertion Sort that introduces *gap-based comparisons*, allowing elements far apart to be compared and swapped before refining to smaller gaps. As the gap decreases, the list becomes increasingly ordered, culminating in a final pass identical to insertion sort.

Implemented gap sequences:

- **Shell:** n/2, n/4, …, 1 (simple, but inefficient for large n).
- **Knuth:** 3h + 1 sequence (improved average-case behavior).
- **Sedgewick:** optimized for large datasets.

Shell Sort is adaptive to partially ordered arrays, but its efficiency depends heavily on the chosen gap sequence.

Reviewer's Algorithm – Heap Sort

Heap Sort uses a *binary heap* to maintain a partially ordered structure where the largest element is always at the root. It proceeds in two steps:

1. **Build Max Heap** – O(n)
2. **Extract Max and Heapify** – n times O(log n)

Heap Sort provides guaranteed **O(n log n)** complexity, independent of data distribution, and sorts in-place.

Conceptual Difference

Aspect	Shell Sort	Heap Sort
Structure	Gap-based insertion	Binary heap tree
Adaptivity	High (nearly sorted arrays perform faster)	None (same cost regardless of order)
Stability	Not stable	Not stable
Implementation	Simple loops with gaps	Tree-based heapify operations

Shell Sort focuses on reducing disorder gradually, while Heap Sort enforces strict order via heap property restoration.

---

2. Complexity Analysis

Time Complexity Comparison

Case	Shell Sort (Knuth)	Heap Sort
Best	O(n log n)	O(n log n)
Average	$O(n^{1\cdot5})$	O(n log n)
Worst	$O(n^{1\cdot5})$	O(n log n)
Space	O(1)	O(1)

Observation:

- Shell Sort can outperform Heap Sort for *small or nearly sorted* datasets due to fewer movements in early passes.
- Heap Sort maintains uniform performance across all cases, making it more predictable for large n.

Mathematical Justification

Shell Sort reduces inversion counts progressively; its performance depends on the number of gap sequences g(n).
Heap Sort builds a heap in linear time:

T(n)=O(n)+n·O(logf_0n)=O(nlogf_0n)T(n)=O(n)+n·O(\log n)=O(n\log n)T(n)=O(n)+n·O(logn)=O(nlogn)

Thus, Heap Sort is *asymptotically superior* for large datasets, but Shell Sort may have smaller constant factors in practice.

---

3. Code Review

Partner's Code (Shell Sort)

- **Strengths:** modular gap-sequence handling, in-place sorting, readable structure.
- **Weaknesses:** redundant iterations for small subarrays, unnecessary temporary variables in swaps.
- **Improvements Suggested:**
  - Use in-place swapping to minimize operations.
  - Skip redundant loops once the array becomes nearly sorted.
  - Prefer Sedgewick sequence for better scaling on large n.

Reviewer's Code (Heap Sort)

- **Strengths:** bottom-up heap construction, consistent O(n log n), scalable to 100 k elements.

- **Weaknesses:** repeated heapify calls even when unnecessary; potential recursion overhead.
- **Optimizations:**
  - Implement iterative heapify.
  - Check if root is already max before swap.
  - Combine with Insertion Sort for small subarrays (hybrid optimization).

Cross Insight:
Both algorithms are in-place and non-stable, but Shell Sort benefits from adaptivity while Heap Sort benefits from mathematical determinism.

---

4. Empirical Results

Array Size	Shell (Knuth) Ops	Heap Sort Ops
1 000	13 884	27 321
5 000	104 020	143 418
10 000	231 453	303 169
100 000	(~ 1.9 M est.)	3 328 065

Analysis:

- Shell Sort exhibits lower raw operation counts on smaller n due to fewer heapify-like rebalancing steps.
- Heap Sort shows *log-linear growth* consistent with O(n log n).
- For large n > 50 000, Heap Sort overtakes Shell Sort in runtime predictability and cache efficiency.

Empirical Validation:

- Theoretical predictions hold: Heap Sort's O(n log n) curve scales steadily.
- Shell Sort's performance varies depending on gap sequence choice and array distribution.

---

5. Conclusion

Both **Shell Sort** and **Heap Sort** are efficient, in-place, comparison-based sorting algorithms.

Criterion	Shell Sort (Knuth)	Heap Sort
Asymptotic Complexity	$O(n^{1.5})$ avg	O(n log n) always
Adaptivity	Good (benefits from order)	None
Memory	O(1)	O(1)
Practical Speed (small n)	Faster	Slightly slower
Scalability (large n)	Degrades	Consistent

Summary of Findings:

- For small or semi-sorted datasets, *Shell Sort* provides quick practical performance.
- For large, random datasets, *Heap Sort* delivers guaranteed and consistent O(n log n) behavior.
- Optimizations in both codes—gap-sequence tuning for Shell, iterative heapify for Heap—can further close the performance gap.

Recommendation:
Hybrid strategies (e.g., Shell Sort → Heap Sort transition or Insertion-Heap hybrids) may leverage both adaptivity and guaranteed efficiency, providing optimal performance across data sizes.